## 1) INTRODUCTION

The next generation of client/server systems will be built inevitably, using distributed objects. They are essentially objects, which can be processed and stored, anywhere on a network. In theory, applications should be able to easily access distributed objects, no matter what they are. This is good, given the growth of client/server computing which will spread from single server departmental Local Area Networks (LAN) to Wide Area Networks (WAN), which will be driven by very high speed, low-cost bandwidth

The problem is that the infrastructure is designed to work only with single tier client server networks. With millions of servers and applications worldwide and the billions of transactions that these systems provide, this is going to be a problem. This is where distributed objects will be able to help.

This paper will discuss the development of distributed objects and their history. It will also discuss the various standards that exist for distributed objects and will compare and contrast them.

---

## 2) WHAT ARE DISTRIBUTED OBJECTS?

Lets start off by describing what an object is in real world terms. It can be defined as a concept, abstraction, or thing with crisp boundaries and meaning for the problem at hand. (Rumbaugh et al, 1991). In software terms, an object is an intelligent piece of a program that can encapsulate code and data. They contain attributes that are its data values. They provide services through methods (also known as operations or functions). Objects provide reuse facilities through inheritance and encapsulation. Reusability can be defined as 'the ability of software products to be reused, in whole or in part, for new applications' (, 1988).

A distributed object, is a piece of code that can live anywhere on a network. They are packaged as independent pieces of code, which can be accessed by remote clients via method invocations. The language and compiler that are used to create distributed objects are transparent to their clients. Clients do not have to know where on the network, a distributed object resides, or what machine it is on, whether it is the same machine or another. It can be executed on a totally different operating system. Distributed objects are therefore able to message each other anywhere in the world.

---

## 3) START OF DISTRIBUTED OBJECTS

The first sort of conceptual idea involving distributed object-oriented technology could be thought of as starting from **Sim++.** Sim++ is a distributed, discrete-event simulation language which uses a theory similar to that of distributed object-oriented systems, in trying to address the resources of a full computer network. Its intention was to model hundreds and thousands of simulation objects on the network. The language tried to emphasise the benefits of addressing distributed execution in an object-oriented manner.

Sim++ is embedded within C++, which shows its object-oriented nature. Described in a paper (Baezner, Lamoe and Unger, 1990), Sim++ uses simulation objects, known as entities, which are independent and loosely coupled. Each entity is an instance of an entity class and will have its own set of state variables. Each entity cannot access the variables or functions of another entity.

Events are data structures, which are created by Sim++ on behalf of the entity, which is scheduling the event. An event will contain attributes, such as the time of the events, its type and a body that is optional. The event time is that simulation at which the event will occur.

When there are more than a couple of entities, which need to communicate within a simulation, the overhead required may reduce the performance overall of the simulation. Sim++ attempts to cure this, by grouping tightly coupled entities into what are known as clusters. Entities within a cluster have to be synchronised with respect to each event scheduled for any entity.

Implementation of a simulation model can involve a trade-off between shared memory speed and paralleled implementation. Within using a distributed implementation of entities, the size of simulation is limited by the amount of memory accessible by the processor. The paper discusses three approaches to a health care simulation. This simulation models a Village and Health Centres as entities and patients as data structures. A wait queue is used for patients who arrive at the health centre for each village, until a health care worker is available.

There is one health centre for each village. Once treated a patient returns to their home village. The diagram, below shows a simplified view of the health care system.

**Figure 1 Health Care System**

The simulation suggests three solutions:

1. Shared memory solution
2. Partially segregated solution
3. Full segregated solution.

1) SHARED MEMORY SOLUTION

This involves a sequential implementation of the model that executes all health centres and villages in a cluster. This is shown in the diagram below:

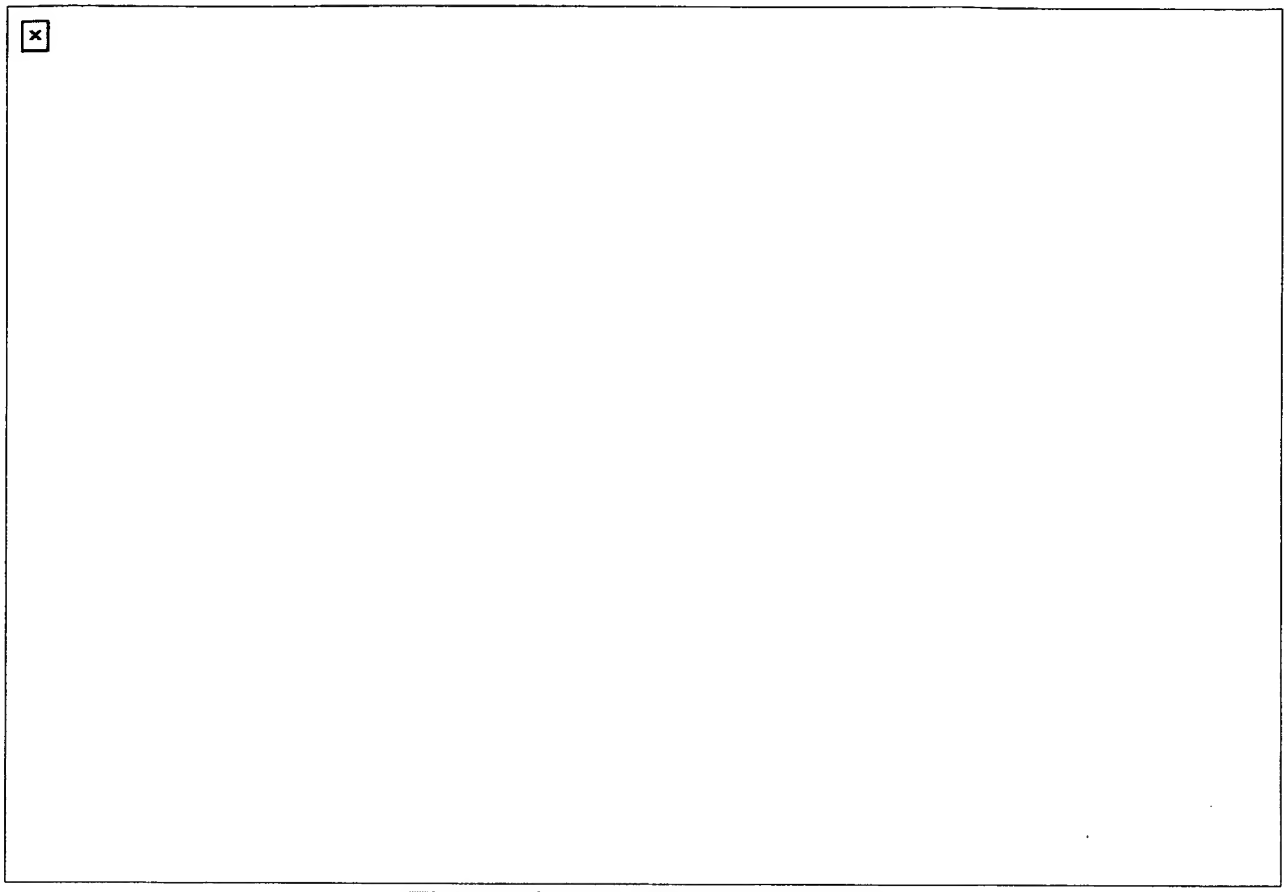

**Figure 2 Shared Memory Solution**

|In this model, all villages and health centres must execute in a single cluster, in order to access each other's member variables and functions. As these entities are being executed on the same single processor, directly accessing the variables of other entities is done but with the overhead of reducing the parallelism desired within execution.

2) PARTIALLY SEGREGATED SOLUTION

This implies putting each village/health centre pair into separate clusters. An event that refers to the patient is scheduled when a patient is referred from one health centre to another. Once the patient has been treated, then an event is scheduled from the health centre that the villager is visiting, back to the village where the villager comes from.

The diagram below illustrates this approach:

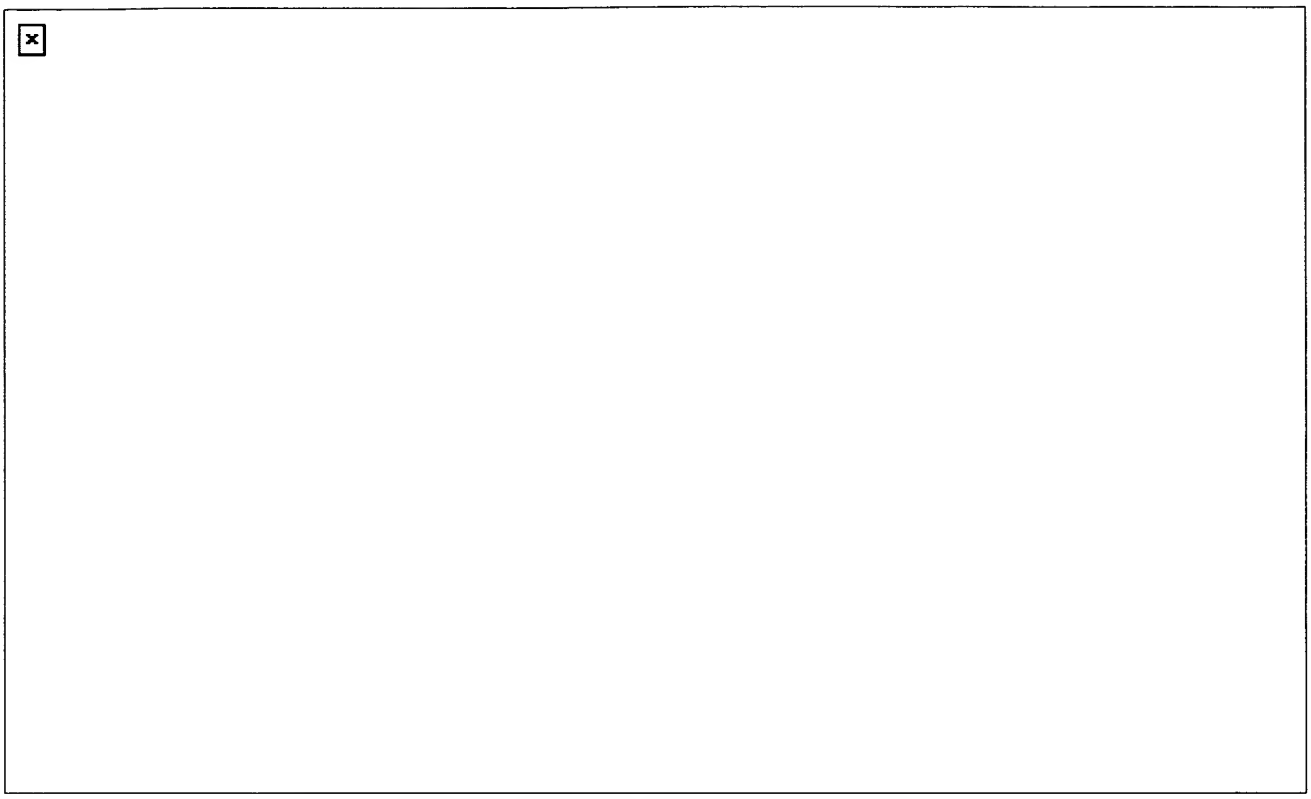

**Figure 3 Partially Segregated Solution**

Shared memory communication is used between a village and its health centre. This approach is more complex than the shared memory or the fully segregated solution, in that entities communicate by means of Events and Shared Memory.

3) FULLY SEGREGATED SOLUTION

This completely separates the address space of entities, so that they are not clustered and cannot access each other's variables and functions. Scheduling and receiving events perform all communication and synchronisation between entities. This method also requires those patients travelling to and from their local health centres are represented by events. The diagram below shows this approach:

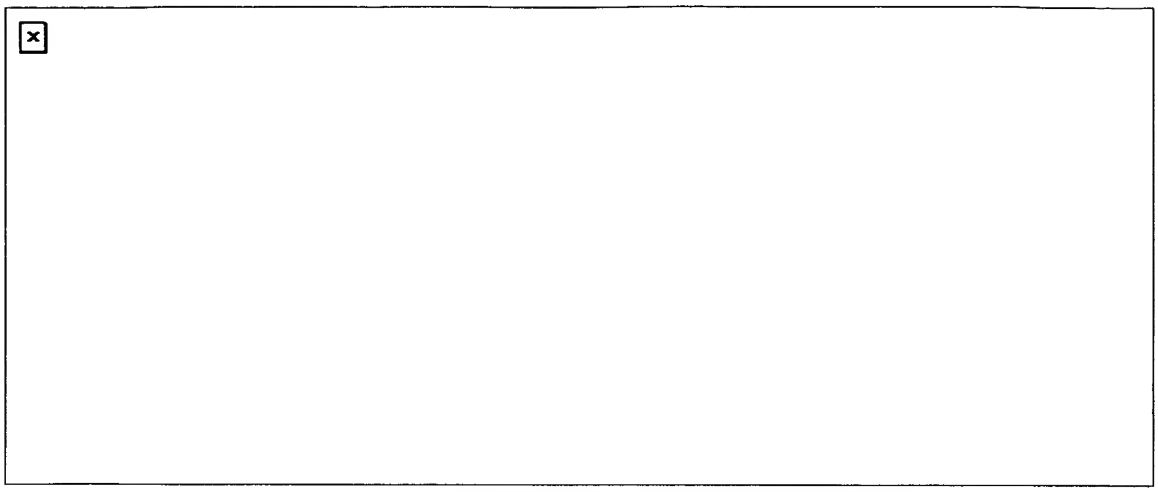

**Figure 4 Fully Segregated Solution**

It seems that out of these approaches, the shared memory solution seems to be effective at sequentially implementing the distribution model. Using a partially segregated solution produces a greater potential for parallelism. Both that and the fully segregated solution, allow a simulation problem to go beyond the scope of a single computer and to a network of distributed computers.

This example shows how distribution can bring benefits of overcoming single computer resources. There have been many other examples of using distributed objects and these will now be discussed, starting with CORBA.

---

**4) CORBA**

CORBA stands for Common Object Request Broker Architecture and was launched in September 1991 by the Object Management Group (OMG). This group was set up in 1989, with the purpose of creating standards allowing for the interoperability and portability of distributed object-oriented applications. The OMG does not actually produce any software, just specifications.

These specifications are created from using ideas and technology provided by OMG members who responds to Requests For Information (RFI) and Requests For Proposals (RFP) which are issued by the OMG.

The Object Management Architecture (OMA) set up by the OMG, attempts to define at a very high level of abstraction, the various facilities necessary for distributed OO computing. At the centre of the OMA is the Object Request Broker (ORB), which is a mechanism that provides transparency of object location, activation and communication. It was this standard; adopted from a joint proposal of Digital Equipment Corporation, Hewlett Packard Company, Hyper Desk Corporation, NCR Corporation and others that formed the first CORBA specification.

CORBA has five main components:

- ORB Core

- Interface Definition Language (IDL)

- Interface Repository

- Dynamic Invocation Interface (DII)

- Object Adapters (OA)

CORBA 1.1 was introduced in 1991, and defined the Interface Definition Language and the Application Programming Interfaces (API), that enable client/server object interaction within a specific implementation of an ORB.

CORBA 2.0 was adopted in December 1994 and this specification defined true interoperability by specifying how ORBs from different vendors can interoperate. In the OMA model, objects provide services and clients issue requests for those services to be performed on their behalf. The ORB is the middleware that establishes the client-server relationships between objects. The client using an ORB, can transparently invoke a method on an object which can be on the same machine or across a network. The ORB intercepts the call and then finds an object, which can implement this request. It then passes the objects parameters, invokes its method and returns the results. The client need not be aware of an object's location, its programming language or operating system. The ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

In developing client-server applications, developers use their own design or define a protocol between devices. Protocols defined will depend on the implementation language, network transport and other factors. ORB's simplify this by defining a protocol through application interfaces via a single implementation independent language specification that is the Interface Definition Language.

Interface Definition Language (IDL), is a declarative language with syntax which is similar to C++. It provides basic data types, constructed types and template types. These are used in operation declarations to define argument types and return types. In turn, operations used in interface declarations define the services provided by objects. The IDL also provides a module construct that can hold interfaces, type definitions and other modules for name scoping purposes.

Interfaces are the most important types provided by the IDL. In addition to describing CORBA objects, they are also used as object reference types. IDL provides interface inheritance, in which derived interfaces inherit the operations and types defined in their basic interfaces.

The Interface Repository, provides persistent storage for IDL interface declarations. The services offered by an IR allow navigation of an object inheritance hierarchy and provide descriptions of all operations that an object supports. Interface Repositories can be used for several purposes. Interface browsers can traverse IR information to help application developers locate potentially reusable software components. The primary function of the IR, is to provide the type information necessary to issue requests using the Dynamic Invocation Interface.

The compilation of IDL declarations into C++ or C stubs, allows clients to invoke operations on known

objects, but some applications must be able to make calls on objects without having compile-time knowledge of their interfaces. A GUI builder might, given a list of object references for drawable components, allow users to browse Irs, learn about the operations supported by each object and then invoke operations on them to see how they display themselves. A GUI builder would only have to know how to traverse IR information and prompt the user for any data necessary to fulfil operation parameter requirements. It could then invoke any operation that the user desires via the Dynamic Invocation Interface (DII).

The DII is a generic client-side stub capable of forwarding any request to any object. It does this by run time interpretation of request parameters and operation identifiers. CORBA allows object implementations to vary widely and in certain cases, a single server program may implement multiple IDL interfaces. In others an IDL interface may be implemented by a variety of shell scripts for each operation. Some applications may have been developed before CORBA was developed or may be OO systems designed specifically to work with an ORB. The ORB provides flexibility to put such legacy applications into a limited implementation criteria, whilst not locking new objects.

An Object Adapter (OA), provides means by which various types of object implementations can use ORB service such as:

- Object Reference Generation

- Object Method Invocation

- Security

- Activation and Deactivation of Objects and Implementations.

An OA may choose to provide these services, by delegating to the ORB or doing the work itself, depending on the underlying ORB.

The components of CORBA fit and work together in the following way. The client invokes an operation supported by an object, provided that it has a valid object reference. The request goes into the IDL surrogates and then to the ORB. The ORB uses the object reference to locate the object implementation and will then pass the request into the Object Adapter managing that object. The OA then feeds the request into the IDL skeleton, which then passes it to the object implementation. Return values generated are then passed through the skeleton and OA to the ORB core. The ORB core returns the values either through the IDL surrogate or the DII to the client application.

---

## 5) COMPONENTS

### 5.1) What Are Components?

Distributed objects are often thought of as components, especially within the OpenDoc and other technologies. Components are standalone objects that can plug and play across network applications, languages, tools and operating systems. Distributed objects are often considered as components because of the way that they are packaged. In distributed object systems, the unit of work and distribution is a component. Distributed object infrastructures make it easy for components to be independent and

autonomous. However not all components are objects, for example an OLE custom control (OCX) is a component which is neither an object nor distributed. Components aim to provide users and developers of software, with the same levels of plug-and-play application interoperability that is available to consumers and manufacturers of electronic parts or custom integrated circuits (Orfali, Harley, Edwards, 1996). There are various leading desktop component standards such as OLE and OpenDoc that use a compound document framework.

### 5.2) Compound Document Framework

Compound documents are metaphors for organising collections of components, both visually and through containment relationships. It is an integration framework for visual components. A document, is a collection site for components and data that can come from a variety of sources. Most users will come into contact with objects through a compound document framework.

### 5.3) OpenDoc

This uses a document metaphor for visually organising components on the desktop. It contains parts as medium-grained objects. Parts can be any type of data such as movies, sound clips, calendars, spreadsheets etc. They are all active concurrently within the same document.

OpenDoc defines rules of engagement for parts to:

1. Seamlessly share screen real estate within a window.
2. Store their data within a single container file.
3. Exchange information with other parts via Links, Clipboards and Drag-and-Drop.
4. Co-ordinate their actions via scripts and semantic events.
5. Interoperate with other desktop component models.

### 5.4) OpenDoc and CORBA

From an user/observer point of CORBA, OpenDoc is merely a CORBA object with desktop smarts. It extends the Object Request Broker part of CORBA to the desktop. Desktop parts can use an ORB to collaborate with other parts and to access server objects wherever they reside. OpenDoc provides component enhancements over and above what CORBA can provide including:

#### i) Reference Counts

OpenDoc maintains counts of who is using its objects. This helps the memory to be refreshed when components are no longer used.

#### ii) Named Extension Suites

Extensions are named suites of services created by components. A client asks a component whether it can support a certain suite of functions and gets a reference to it. Extensions in OpenDoc form the basis for collaborative desktops consisting of "very smart" parts that act like "vertical", "horizontal" or client/server suites.

iii) Property Editors

OpenDoc provides dialog boxes or notebooks through GUI interfaces for the purpose of editing the properties of a component and scripts at run time. This allows investigation and modification of components by users and integrators.

iv) Semantic Messaging

OpenDocs Open Systems Architecture (OSA) complements the messaging model of CORBA, by introducing a set of verbs and object specifiers that let clients manipulate server parts at the semantic level.

Semantic messages contain object specifiers that describe in human terms, what the user can see. The message system resolves these descriptions into target objects with unique references.

v) Scriptable Components

OpenDoc parts are scriptable, allowing users and system integrators to quickly assemble groups of components and integrate them using scripts. These make it easy to combine components from different sources into custom solutions.

---

**6) OLE**

OLE is another type of compound document technology that offers more than its competitors. It stands for Object Linking and Embedding and is a complete environment for components, which can match CORBA and OpenDoc, in terms of functionality. Microsoft developed it as a basis for their future distributed computing environments in 1990, using a compound document framework.

OLE uses a set of common services, which allow components to collaborate intelligently. Like CORBA , it covers the entire spectrum concerning components from fine-grained objects to coarse-grained existing applications. A class that implements one or more interfaces, as a class factory defines an OLE component. This is an interface, which knows how to produce a component instance of that class.

OLE components unlike, OpenDoc parts, are groups of interfaces. It would require many interfaces to make an OLE component as detailed as an OpenDoc part. OLE, unlike CORBA however does not address the current needs of distributed components. COM provides the first step towards distributed objects.

---

## 7) COM

COM (Component Object Model) specifies interfaces between component objects within a single or between multiple applications. Like CORBA, it separates the interface from the implementation. Also COM requires that all shared code be declared by using object interfaces.

Its components can support multiple interfaces but its object model cannot support multiple inheritance. COM defines interfaces for object factories and provides a rudimentary component licensing mechanism. A COM client asks for the services of a component by passing a unique class identifier (CLSID)

---

## 8) CONCLUSION

COM and CORBA essentially present problems in terms of interoperability , as they have dissimilar object models and therefore their components will never be able to fully collaborate across object environments.

OMG have been trying to figure two standards between CORBA and COM., so that one targets local COM and a later one that targets a distributed COM. A paper (Rochelau, 1996) proposed that COM and CORBA may be able to be linked in order to specify bi-directional communication between CORBA objects and COM objects. The goal of this is that objects from one model would be able to be viewed in another and key functionality would be visible to clients using another system.

OpenDoc and OLE currently lead the way in component standards for the desktop. The OMG is also developing an OLE/CORBA spec that would standardise the way an operation could be carried out on a CORBA object from an OLE application.

These significant developments suggest that despite the various competing standards that have emerged, distributed objects have a viable and effective future in the world of client-server computing. Possible insights into the future suggest extensive involvement of the Internet through the development of ActiveX by Microsoft.

---

## BIBLIOGRAPHY

1) Orfali, R, Harkey, D., Edwards, J The Essential Distributed Objects Survival Guide, 1996, John Wily and Sons , Canada

2) Meyer, B Object-Oriented Software Construction, 1988, Prentice Hall.

3) Soley, RM et al Object Management Archictecture Guide, 1995, OMG, Canada.

4) Baezner D, Lomow, G , Unger BW, "Sim++: The Transition to distributes simulation", Proceedings of the SCS Mutilconference on Distributed Simulation, VOL 22, No 2, p211-18, January 1990.

5) Rochelau, C " Users Gain COM/CORBA Interoperability, OMG, 1996